Automatic Differentiation of Third Order Derivatives using Forward Mode in C++

Master Thesis
Department of Informatics
University of Bergen

Torbjørn Lium

January 7, 2009

Contents

Chapter 1

Introduction

The concept of the derivative is at the core of calculus and modern mathematics, and is consequently also at the core of many fields of science. The derivatives can of course be calculated by hand, but even for problems of modest size this is a time-consuming task which is prone to errors. Divided differences can also be used to calculate derivatives, but truncating errors may lead to significant errors in the derivatives as well as cancellation errors.

Automatic differentiation avoids all of the drawbacks associated with the techniques above. The main drawback with symbolic differentiation is low speed, and the difficulty of converting a computer program into a single expression. Moreover, many functions grow quite complex as higher derivatives are calculated. Two important drawbacks with finite differences are round-off errors in the discretization process and cancellation. Both classical methods have problems with calculating higher derivatives, where the complexity and errors increase.

Automatic differentiation exploits the fact that any computer program that implements a vector function can generally be decomposed into a sequence of elementary assignments. The elemental partial derivatives are combined in accordance with the chain rule from calculus. This can be used to form some derivative informations for the function such as gradients, Jacobians, Hessians.

1.1 Literature and tools

The concept of automatic differentiation has been invented and re-invented independently by many authors. A very comprehensive and authorative reference on automatic differentiation is the book of Griewank [5]. A very good starting point for information on the web would be the website *autodiff.org*,

a free and non-commercial community portal to automatic differentiation available to researchers with a standing interest in AD and its applications.

Automatic differentiation is also starting to appear in literature which does not specifically deal with automatic differentiation, such as [15]. Automatic differentiation is also covered in [1], calling its implementation of multicomponent objects *Rall numbers* and naming Louis B. Rall "the mathematician who developed the automatic differentiation technique for computers [19]".

A number of good software tools appeared as the field of automatic differentiation grew, particularly during the 1990s. ADIFOR [3], ADOL-C [6], and TAPENADE [17] are but a few of the many tools freely available. Commercial tools are also available, such as those from FastOpt.

1.2 Implementing automatic differentiation

Automatic differentiation is a technique for augmenting computer programs with derivative computations. It exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations such as additions or elementary functions such as exp(). By applying the chain rule of derivative calculus repeatedly to these operations, derivatives of arbitrary order can be computed automatically, and accurate to working precision.

Traditionally, two approaches to automatic differentiation have been developed: the so-called forward and reverse modes. Forward accumulation traverse the chain rule of calculus from right to left, while reverse accumulation traverse the chain rule from left to right. These two modes of automatic differentiation are just two extreme ways of traversing the chain rule, and an implementation may incorporate both forward and reverse into a hybrid mode. The problem of computing the full Jacobian of $F : \mathbb{R}^n \to \mathbb{R}^m$ with a minimum number of arithmetic operations is proven to be NP-complete [14].

Automatic differentiation can be implemented using several techniques depending on the requirements of the implementation. Generally, one of two strategies are used: source transformation or operator overloading.

With source transformation, a function is replaced by automatically generated source code that includes statements for calculating the derivatives. This code is interleaved within the original instructions. Source transformation can be implemented for all programming languages, and it might make compile time optimizations easier. However, the implementation of the automatic differentiation tool itself is more difficult. Tools such as ADIFOR and TAPENADE performs source transformation of Fortran programs, the

latter also on programs written in C.

Operator overloading is another possibility for source code written in a language that supports it, such as C, C++ or Fortran, where operator overloading was introduced in Fortran90. With operator overloading, objects for real numbers and mathematical operations are overloaded to cater for an augmented arithmetic. There are several implementations of automatic differentiation using operator overloading freely available, two of which are ADOL-C [6] and FADBAD++ [2].

In this thesis we will focus on the forward mode of automatic differentiation, implemented using operator overloading in C++. Forward mode automatic differentiation is accomplished by augmenting the algebra of real numbers and obtaining a new arithmetic. This new arithmetic consists of ordered pairs (x, x') with ordinary arithmetic on the first component, and first order differentiation arithmetic on the second component. We will extend this arithmetic to calculate the first, second, and third derivatives automatically. While we in this thesis will use optimization as the field of choice for a practical application, using third order derivatives is also interesting in other fields, such as sensitivity analysis [10].

1.3 Choice of programming language

C++ is a language with many features which make it attractive for scientific computing. Templates for generic programming, operator overloading and object-orientation are but a few of the many useful features of C++. The concepts of operator overloading and object-orientation are of particular interest when we want to implement automatic differentiation. A variety of compilers and supporting libraries are also freely available.

Despite these features, the scientific computing community has been reluctant to adopt C++, partly due to performance problems. In the early 1990s, C++ programs were much slower than their Fortran counterparts, but the performance of C++ programs has improved markedly due to a combination of better optimizing compilers and new library techniques [23].

Even though memory management was not a deciding factor in the choice of programming language for this thesis, a very important feature of a language is how it handles memory. Many languages, such as Java and C#, provide automated memory handling, or garbage collection. This can save a lot of time for the developers by automatically deciding when memory is ready to reuse, rather than the programmer having to write code to do it. C++ is one of the languages that require the programmer to handle memory allocation and deallocation explicitly. While this provides the skilled

programmer with a very high level of control over how the program uses memory, loosing track of said memory tends to cause all sorts of bugs which are hard to solve and slow down further development.

Chapter 2

Operator Overloading

Operator overloading provides an intuitive interface to our code. It allows C/C++ operators to have user-defined meanings based on user-defined types. The C++ language itself overloads several operators, for instance the operator «. This operator is built into C++ and can be used both as as the stream insertion operator as well as the the bitwise left shift operator. The » operator is overloaded similarly for stream output and bitwise right shift. Both of these operators are overloaded in the C++ Standard Library. Operators like + and - perform differently depending on their context in integer, floating point or pointer arithmetic. Function calls may of course perform the same jobs as overloaded operators, but as we will soon find out, the operator notation is often clearer.

The names of, precedence of, associativity of, and arity of operators is fixed by the language. We will not get any additional functionality to the code by using operator overloading, and it will still compile to function calls. Operators may be overloaded by writing a non-static member-function or a global function where the function name is the keyword *operator* followed by the symbol of the operator being overloaded. Operators overloaded as member functions must be non-static because they must be called on an object of the class and operate on that object.
It is worth mentioning that it is not possible to create new operators. Only operators existing in the C++ language can be overloaded. Neither is it possible to alter the meaning of how an operator works on fundamental types.

There are a few possible pitfalls in using operator overloading in your code. The most important part to have in mind is not to confuse the users of your code. It might be tempting to overload the ^-operator to work for to-the-power-of, except it has the wrong precedence and associativity.

You would be much better off by overloading pow(base, exponent), a double precision version of which is in the <cmath> library.
Most operators in C++ can be overloaded. Operators . and *?:* are the only two C operators which can not be overloaded. Neither can sizeof, which is technically an operator. C++ adds a few of its own operators, most of which can be overloaded except *::* and *.*.

2.1 Complex numbers

Using complex numbers is a very good example to illustrate the intuitive interface operator overloading will provide for the users of our code. Complex numbers are often written on the form $a+bi$ where a and b are real numbers and i the imaginary unit with the property $i^2 = -1$. It is also possible to represent complex numbers as ordered pairs of real numbers. The complex number $a + bi$ corresponds to the unique ordered pair (a, b). It is easy to imagine ordered pairs of real numbers as its own class where constructors handle the initial value of the ordered pair. The algebraic operations on these ordered pairs are also easily implemented from following example formulas:

$$(a, b) + (c, d) = \quad (a + c, b + d) \tag{2.1}$$
$$(a, b) * (c, d) = \quad (ac - bd, bc + ad) \tag{2.2}$$

The following example will show how to overload c++ operators to deal with the addition and multiplication of complex numbers defined in its own class. Object oriented programming, function inlining and other language specific topics will not be discussed. These topics are of course important for implementation details and performance, but we will keep focus on the operators themselves throughout this thesis.

```
1  #include <iostream>
2
3  class Complex {
4    public:
5    // A couple of constructors.
6    Complex() { real = 0; imag = 0;}
7    Complex(double r; double i) { real = r; imag = i;}
8    // get/set and other member functions omitted
9    ...
10   // Operators
11   Complex operator + (const Complex& x);
12   friend operator - (const double& x, const Complex& y);
13   Complex operator * (const Complex& x);
14   Complex operator * (const double& x);
15   Complex operator / (const Complex& x);
```

```
16
17    private:
18    double real; // Real part
19    double imag; // Imaginary part
20 };
21
22 Complex::Complex operator + (const Complex& x) {
23    Complex temp();
24    double r = real + x.real;
25    double i = imag + x.imag;
26    temp.setValues(r,i);
27    return temp;
28 }
29
30 Complex::Complex operator * (const Complex& x) {
31    Complex temp();
32    double r = real*x.real - imag*x.imag;
33    double i = imag*x.real + real*x.imag;
34    temp.setValues(r,i);
35    return temp;
36 }
```

Listing 2.1: Excerpt of Complex.h

As the listing above will show, it is not especially difficult to start using operator overloading. The jobs performed by the operators are easily understood and the end result is a class which is easy to use. Now that we know how to defined our class of complex numbers along with the necessary operators, we can very quickly begin doing some interesting computation.

One function which should be familiar to anyone who has studied calculus is the Newton-Raphson iteration. It is a method for finding roots of a function by using the derivative of the function to improve on an approximation to the root. To perform a Newton-Raphson approximation, we will need a function $f(x)$, its derivative $f'(x)$ and a an approximation to a root of the function. The Newton-Raphson procedure works by calculating a closer approximation to the root by calculating $x' = x - \frac{f(x)}{f'(x)}$. This procedure will typically be repeated until the successive values are close together. When the difference between these values are within a certain tolerance we conclude that we have a very good approximation to the actual value $x^\star$ for which $f(x^\star) = 0$.

Consider the function

$$f(z) = z^3 - 1 \tag{2.3}$$

and its derivative

$$f'(z) = 3z^2 \tag{2.4}$$

where z is a complex number. This function has three roots in the complex plane. The first root is located at the point $(1, 0)$, the second at $(-0.5, -\sin(\frac{\pi}{3}))$ and the third at $(-0.5, \sin(\frac{\pi}{3}))$. To be able to use this function we need the multiplication, division and subtraction operators of our complex number class to be overloaded. The updated approximations will be calculated by

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)} \tag{2.5}$$

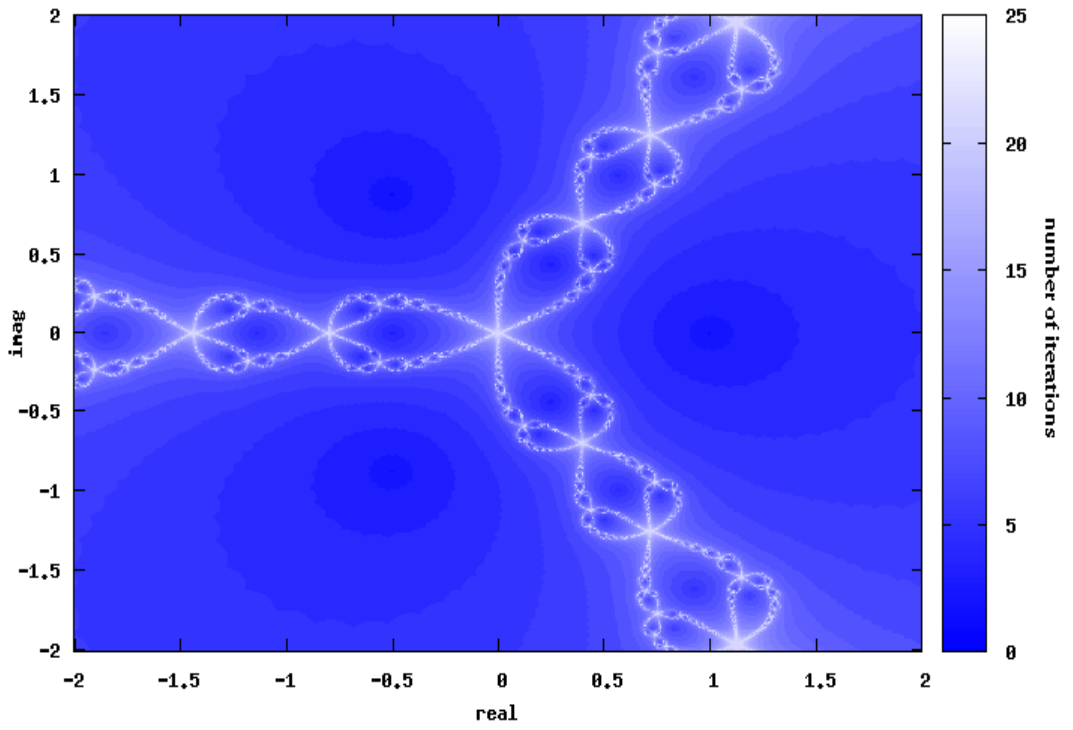
where the initial guess z_0 is a point in the complex plane. Assuming our class of complex numbers implements these operators, consider the following

```
1  #include <iostream>
2  #include <fstream>
3  #include "math.h"
4  #include "Complex.h"
5
6  int main () {
7    const double sin60 = 0.866025; //Constant variable.
8    const double cos60 = 0.5; //Constant variable.
9    const double eps = 1.0e-4; //Convergence tolerance
10   const int maxit = 20; //Maximum number of iterations
11   ofstream fil;
12   fil.open("newton.dat"); // Data file for our plot points.
13   Complex z;
14   double zr;
15   double zi;
16   int it;
17   for(double r=2; r>-2; r-=0.001) {
18     for(double i=2; i>-2; i-=0.001) {
19       z.setValues(r,i);
20       it = 0;
21       while(it<) {
22         zr = z.getReal();
23         zi = z.getImag();
24
25         if(fabs(zr-1)<eps && fabs(zi) < eps)
26           break; // converged to root (1,0)
27
28         else if(fabs(zr+cos60)<eps && fabs(zi+sin60) < eps)
29           break; //converged to root (-0.5,-0.866025)
30
```

```
31            else if(fabs(zr+cos60)<eps && fabs(zi-sin60) < eps)
32              break; //converged to root (-0.5,0.866025)
33
34            z = z - (z*z*z-1)/(3*z*z);
35            it++;
36          }
37          fil << r << " " << i << " " << it << endl;
38        }
39      }
40      fil.close();
41      return 0;
42  }
```

Listing 2.2: Using our complex number class

The above listing 2.2 will traverse the complex plane from -2 to 2 on both axis. It will use the points in the plane as the starting guess of a root of $f(x) = z^3 - 1$. Then it performs the Newton-Raphson iteration and updates the guess of a root until one of the three roots are found or the maximum number of iterations is reached. When either of the stopping criteria is met, the point along with the counted number of iterations is then printed to a file. The following image is made by using Gnuplot to plot the data contained in our data file produced by listing 2.2.



The colours in the plot above runs from dark blue to white signifying

the number of Newton-Raphson iterations needed to reach one of the three roots of (2.3). The three roots can be seen as the darks spots at $(1, 0)$, $(-0.5, -\sin(\frac{\pi}{3}))$, and $(-0.5, \sin(\frac{\pi}{3}))$.

2.2 The ADouble class

Now that we have covered basics of operator overloading we can begin to produce some code to calculate our derivatives in earnest. We will start coding a new class of numbers called ADouble. For now we will only consider univariate functions such as

$$f(x) = x^2 + 3x + 4$$

Just as we did with complex numbers, our class will at its core consist of the ordered pair (u, u') where both u and u' are of type double. In our ordered pair u will represent the variable's real value and u' will represent the variable's derivative value. This derivative value will be initialized to 1. The key to calculate our derivatives lies in the fact that any computer program that implements a function can (generally) be decomposed into a sequence of elementary assignments. These elemental partial derivatives are combined in accordance with the rules of calculus to form the derivative information, such as that done in [19] to provide a convenient list of first and second differentials.

Consider the function

$$f(x) = x \cdot x$$

Using Leibniz notation the product rule may be stated as

$$\frac{d}{dx}(u \cdot v) = u\frac{dv}{dx} + v\frac{du}{dx}$$

which can also be written in 'prime notation' as

$$(u \cdot v)' = u' \cdot v + u \cdot v'. \tag{2.6}$$

If u and v are implemented as the ordered pair we will use in our ADouble class, only the operator needs to be overloaded to be able to produce the derivative of the function. Now consider the following header file for our ADouble class.

```
1  #ifndef ADOUBLE_H_
2  #define ADOUBLE_H_
3
4  class ADouble {
5  private:
6    double value;
7    double deriv;
8
9  public:
10   ADouble(double v);
11
12   ADouble operator* (const ADouble& x);
13   ADouble& operator= (const ADouble& x);
14
15   double getVal() { return value; }
16   double getDer() { return deriv; }
17 };
18 #endif // ADOUBLE_H_
```

Listing 2.3: Header file for our ADouble class

The class data is nothing more than two doubles. One double representing the value and the other representing the derivative value. A simple constructor is provided to intitialize the ADouble variable to a certain (real) value. The derivative value is initialized to 1 because the derivative of the single variable x is $\frac{d}{dx}x = 1$.

In the source file for our ADouble class we will define the multiplication operator to work in accordance with the product rule as stated in (2.6). A simple assignment operator is also provided.

```
1  #include <iostream>
2  #include <iomanip>
3  #include "ADouble.h"
4
5  ADouble::ADouble(double v) {
6    value = v;
7    deriv = 1;
8  }
9
10 ADouble ADouble::operator*(const ADouble& x) {
11   ADouble temp(0);
12   temp.value = value * x.value;
13   temp.deriv = deriv * x.value + value * x.deriv;
14   return temp;
15 }
16
17 ADouble& ADouble::operator= (const ADouble& x) {
```

```
18    value = x.value;
19    deriv = x.deriv;
20    return *this;
21  }
```

Listing 2.4: Source file of our ADouble class

And that is all we need to start calculating some derivatives, at least as far as simple multiplication goes. By using operator overloading we are able to code our functions to be strikingly similar to their mathematical definitions. now consider the very simple function

$$f(x) = x \cdot x \cdot x$$

We will choose the independent variable x to have the value 2.0. The function value at $x = 2$ will be $f(2) = 8$ and its derivative will be $f'(2) = 12$.

```
1  #include <iostream>
2  #include "ADouble.h"
3
4  int main() {
5    ADouble x(2.0); //Initialize x to 2.0
6    ADouble f = x * x * x;
7    std::cout << "f(x) = x * x * x\n";
8    std::cout << "x=2.0\n";
9    std::cout << "f(x) = " << f.getVal() << std::endl;
10   std::cout << "f'(x) = " << f.getDer() << std::endl;
11   return 0;
12 }
```

Listing 2.5: The first small steps of automatic differentiation

This very simple example will produce the following output:

```
f(x) = x * x * x
x = 2.0
f(x) = 8
f'(x) = 12
```

which is of course exactly what we expected. The function value along with its derivative at the given point is calculated without any special commands or function calls.

Chapter 3

Automatic Differentiation of multivariate functions

3.1 Extending ADouble to multivariate functions

We will start our discussion of automatic differentiation of multivariate functions much in the same vein as done in [13] to extend our ADouble.

The first derivative of a scalar function $f(x)$ with respect the vector variable $x = (x_0, \ldots, x_{n-1})$ is called the *gradient* and is denoted by ∇f where the symbol ∇ denotes the vector differential operator. The gradient of f is defined to be the vector field whose components are the partial derivatives of f. That is:

$$\nabla f = \left(\frac{\partial f}{\partial x_0}, \ldots, \frac{\partial f}{\partial x_{n-1}} \right)^T.$$

Consider the function

$$f(x_0, x_1) = u * v = u(x_0, x_1) * v(x_0, x_1)$$

Let the first partial derivative of f with respect to x_i be designated f_i. The rules of calculus give the following for the value and first derivative of f:

$$\begin{aligned} f &= u * v \\ f_i &= u_i * v + u * v_i \end{aligned}$$

The designation of the first partial derivative of f to be f_i is made to emphasise the similarity of arrays and partial derivatives. It is clear that if the

values of u, v and their partial derivatives is known then the value of f and its partial derivatives can be calculated by the relationships above. Similar relationships for the operations of addition, subtraction and division are produced by the rules of calculus.

We are interested in finding the values of the derivative at a point of the function $f(x)$ where x is the vector in $\mathbb{R}^n$. Because the value of this vector is known, as well as its partial derivatives, the value and partial derivatives of f is known.

In the following listing we will expand our ADouble class to work on the first derivative of the multivariate function $f(x)$

```
1  #ifndef ADOUBLE_H_
2  #define ADOUBLE_H_
3
4  class ADouble {
5  private:
6    double value;
7    double * gradient;
8    int num_vars;
9
10 public:
11   ADouble(int num_vars, double v, int diffindex);
12   virtual ~ADouble();
13
14   ADouble operator* (const ADouble& x);
15   ADouble& operator= (const ADouble& x);
16   friend std::ostream& operator<< (std::ostream& s, const ADouble
         &);
17 };
18 #endif // ADOUBLE_H_
```

Listing 3.1: Header definition of our modified ADouble class

Instead of using a single variable of type double to handle the derivative value we now use a dynamically allocated array of doubles. This array is 0-indexed. We also need to keep track of how many variables in the function. The constructor is also modified. Instead of just taking the one argument as we did in listing 2.3 we now take as arguments the number of variables, the (real) value and *diffindex*. This *diffindex* denotes the index number in the gradient vector.
Because we dynamically allocate the memory needed for the gradient, we also need the destructor on line 12. The definition of the multiplication operator and assignment operator is exactly the same in Listing 3.1 as it was in

Listing 2.3. The last change is the addition of the overloaded output stream operator «.

In the following listing we will modify the source file of our ADouble class to work with multivariate functions.

```
1  #include <iostream>
2  #include "ADouble1.h"
3
4  ADouble::ADouble(int nvars, double v, int diffindex) {
5    value = v;
6    num_vars = nvars;
7    gradient = new double[num_vars];
8    for(int i=0; i<num_vars; i++) {
9      gradient[i] = 0;
10   }
11   gradient[diffindex] = 1;
12 }
13
14 ADouble::~ADouble() {
15   delete [] gradient;
16 }
17
18 ADouble ADouble::operator*(const ADouble& x) {
19   ADouble temp(x.num_vars, 0.0, 0);
20   temp.value = value * x.value;
21   for(int i=0; i<x.num_vars; i++) {
22     temp.gradient[i] = (gradient[i] * x.value) + (value * x.
         gradient[i]);
23   }
24   return temp;
25 }
26
27 ADouble& ADouble::operator= (const ADouble& x) {
28   value = x.value;
29   for(int i=0; i<num_vars; i++) {
30     gradient[i] = x.gradient[i];
31   }
32   return *this;
33 }
34
35 std::ostream& operator<< (std::ostream& s, const ADouble& x) {
36   s << "Value: " << x.value << std::endl;
37   s << "Gradient: [";
38   for(int i=0; i<x.num_vars; i++) {
39     s << " " << x.gradient[i] << " ";
40   }
```

```
41    s << "]" << std::endl;
42    return s;
43  }
```

Listing 3.2: Source file of our ADouble class

The constructor in Listing 3.2 will still assign the variable's real value in the same manner as in Listing 2.4, but the initialisation of the gradient is not quite as obvious. We will still initialise the derivative of a variable to be 1, but only at the correct index. The variable x_1 in the vector (x_0, x_1, x_2) will of course be of type ADouble, but its initial gradient will be $(0, 1, 0)$ because x_1 is not differentiable with respect to to x_0 or x_2.
Further, the multiplication operator is modified to do the proper operations on the gradient.

Using our modified ADouble class is no more complicated than it was in Listing 2.5. As an example we what to calculate the function value of

$$f(x, y, z) = x * y * z$$

and its gradient

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)^T$$
$$\nabla f(x, y, z) = (1 \cdot y \cdot z, x \cdot 1 \cdot z, x \cdot y \cdot 1)^T$$

at the point

$$(x, y, z)^T = (2.0, 3.0, 4.0)^T$$

The function value should be

$$f(x, y, z) = 2 \cdot 3 \cdot 4 = 24$$

and the gradient should be

$$\nabla f(x, y, z) = (1 \cdot 3 \cdot 4, 2 \cdot 1 \cdot 4, 2 \cdot 3 \cdot 1)^T = (12, 8, 6)^T$$

Consider the following short code example of how we now use our ADouble class in our code

```
1 #include <iostream>
2 #include "ADouble1.h"
3
4 int main() {
5   const int num_vars = 3;
6   ADouble x(num_vars, 2.0, 0);
```

```
 7    ADouble y(num_vars, 3.0, 1);
 8    ADouble z(num_vars, 4.0, 2);
 9    ADouble f = x * y * z;
10    std::cout << f;
11    return 0;
12  }
```

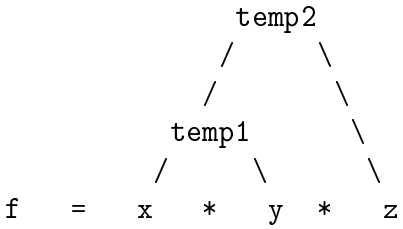
Listing 3.3: Using our our ADouble class

First of all we need to declare how many variables our function consists of. The new constructor for the ADouble class is called to initialise variables x, y and z to the desired values. The function value f is also initialized as an instance of the ADouble class. We also overloaded the output stream operator to provide a convenient way of printing our resulting ADouble object f to standard output. The resulting output from Listing 3.3 is

```
Value: 24
Gradient: [ 12  8  6 ]
```

Again exactly what we want.

What may not be so clear is how the gradient is filled with the correct numbers. Consider the simple diagram below:



Within then independent variable x is the temporary gradient $(1, 0, 0)$. The independent variables y and z have the gradients $(0, 1, 0)$ and $(0, 0, 1)$ respectively. At the first multiplication, a temporary object of type ADouble is created, designated *temp1* in the diagram above, which contains the function value and gradient with respect to x and y. This temporary object is then multiplied with z to produce the second temporary object *temp2*. This latest temporary object contains the full function value and gradient with respect to x, y and z. Finally, f is set equal to the last temporary object, and thus contains the function value and gradient of the entire function.

Chapter 4

Extending our AD-class to the second derivative and higher

4.1 Matrices and second order partial derivatives

Before we start doing anything more with our ADouble class, a few words on matrices and second order partial derivatives is needed. The reason for this will soon become clear.

A symmetric matrix is a square matrix which has the property that the matrix A is equal to its transpose.

$$A = A^T$$

Each entry in a symmetric matrix is symmetric with respect to its main diagonal. If the entries of the matrix A is written as a_{ij}, then $a_{ij} = a_{ji}$ for all indices i and j. The following 3 x 3 matrix is symmetric

$$\begin{bmatrix} 2 & 1 & -3 \\ 1 & 4 & 8 \\ -3 & 8 & 6 \end{bmatrix}$$

The *Hessian* matrix is the n x n matrix of second order partial derivatives of a function. It describes the local curvature of a function of many variables. It was developed by the German mathematician Ludwig Otto Hesse in the 19th century and later named after him. We will use the symbol $\nabla^2 f$ to denote the Hessian matrix of the multivariate function f.

Given the two times continuously differentiable function

$$f(x)$$

where x is the vector in $\mathbb{R}^n$, the Hessian matrix is on the form

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_0^2} & \frac{\partial^2 f}{\partial x_0 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_0 \partial x_{n-1}} \\ \frac{\partial^2 f}{\partial x_1 \partial x_0} & \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_{n-1} \partial x_0} & \frac{\partial^2 f}{\partial x_{n-1} \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_{n-1}^2} \end{bmatrix}$$

The order in which the partial derivatives is produced does not matter. If the function $f\colon \mathbb{R}^n \to \mathbb{R}$ has continuous partial derivatives at any given point in $\mathbb{R}^n$, then for $0 \leq i, j < n$

$$\frac{\partial^2 f}{\partial x_i \, \partial x_j} x = \frac{\partial^2 f}{\partial x_j \, \partial x_i} x \tag{4.1}$$

In other words, the partial derivatives of this function commute at that point and the Hessian matrix is symmetric. This enables us to store and do calculations on merely one of the halves of the Hessian along with its main diagonal.

At this point we begin to see a worrying development regarding computational complexity and storage requirements. If we continue along the path we have so far, the storage requirements will grow very large very quickly. The gradient itself is not really a problem with its n required doubles, but the Hessian will soon grow to be too large due to its required $\frac{n(n+1)}{2}$ doubles to store. As we shall soon find out, the storage requirement the third partial derivatives is even worse, but for the time being we will ignore this bottleneck as the code for a somewhat naive implementation is more easily understood.

4.2 Extending ADouble to second partial derivatives

In order to actually use any of the information above, we will need some rules to base our operators on. To keep the displayed amount of example code to

a minimum we will only consider the multiplication operator.

Consider the function

$$f = u \cdot v$$

Component i of the gradient is then

$$f_i = \frac{\partial}{\partial x_i} u \cdot v = u_i \cdot v + u \cdot v_i$$

Element ij of the Hessian is calculated using

$$f_{ij} = u_{ij} \cdot v + u_i \cdot v_j + u_j \cdot v_i + u \cdot v_{ij} \tag{4.2}$$

We are now in the position to start making modifications to our ADouble class. Our goal now is to successfully compute the (full) Hessian matrix. Consider the following header file for our modified ADouble class.

```
1  #ifndef ADOUBLE_H_
2  #define ADOUBLE_H_
3
4  class ADouble {
5  private:
6    double value;
7    double * gradient;
8    double ** Hessian;
9    int num_vars;
10
11 public:
12   ADouble(int num_vars, double v, int diffindex);
13   virtual ~ADouble();
14
15   ADouble operator+ (const ADouble& x);
16   ADouble operator* (const ADouble& x);
17   ADouble& operator= (const ADouble& x);
18   friend std::ostream& operator<< (std::ostream& s, const ADouble
         &);
19 };
20 #endif // ADOUBLE_H_
```

Listing 4.1: Header file of our ADouble class

Only two things has changed compared to Listing 3.1. We have added the *Hessian* declaration on line 8. Here the Hessian is defined as a dynamically allocated two dimensional array. We have also introduced the addition operator. Note that nothing more has been changed. The other two operators still take the same arguments. We also need to do some changes in the source

file for our ADouble class. Rather than Listing the whole source file, only the source for the class constructor and the multiplication operator will be listed in its entirety.

First we need to allocate the memory needed to store the Hessian. This will be done in the constructor as well as initialising the Hessian to 0 at all its elements. We also need to modify our multiplication operator to calculate the elements of the Hessian according to (4.2).

Consider the new ADouble class constructor

```
1  ADouble::ADouble(int nvars, double v, int diffindex) {
2    value = v;
3    num_vars = nvars;
4    gradient = new double[num_vars];
5    for(int i=0; i<num_vars; i++) {
6      gradient[i] = 0;
7    }
8    gradient[diffindex] = 1;
9    Hessian = new double*[num_vars];
10   for(int i=0; i<num_vars; i++) {
11     Hessian[i] = new double[i+1];
12   }
13   for(int i=0; i<num_vars; i++) {
14     for(int j=0; j<=i; j++) {
15       Hessian[i][j] = 0;
16     }
17   }
18 }
```

Listing 4.2: Our modified ADouble class constructor

Our constructor has seen some radical changes since Listing 3.2 with new code from lines 9 through 16. Because we want to be able to decide the number of variables at runtime, a dynamic two dimensional array is needed to store the Hessian. Or more accurately, an array of arrays is needed. We also take the symmetry of the Hessian into account by storing only the bottom half of the matrix along with its main diagonal.

As we continue to allocate memory, we need to keep an eye out for so-called memory leaks. If we allocate more memory than we free up after we have used it, the computer can run out of available memory. Every time we use the operators *new* and *new*[] in our code, we need to use the corresponding *delete* and *delete*[] to release the memory at the proper place. The proper place is our ADouble class destructor, as show in the following Listing.

```
1 ADouble::~ADouble() {
2   delete [] gradient;
3   for(int i=0; i<num_vars; i++) {
4     delete [] Hessian[i];
5   }
6   delete [] Hessian;
7 }
```

Listing 4.3: A destructor for the ADouble class

The destructor is not the most sophisticated piece of code, but it does the job we need it to. Namely to free the memory used by the gradient and the Hessian.

We will also need to make some changes to our multiplication operator in order to actually perform the operations required to fill our Hessian matrix. The following Listing will provide for that

```
1  ADouble ADouble::operator*(const ADouble& x) {
2    ADouble temp(x.num_vars, 0.0, 0);
3    temp.value = value * x.value;
4    for(int i=0; i<x.num_vars; i++) {
5      temp.gradient[i] = (gradient[i] * x.value) +
6        (value * x.gradient[i]);
7    }
8    for(int i=0; i<x.num_vars; i++) {
9      for(int j=0; j<=i; j++) {
10       temp.Hessian[i][j] =
11         Hessian[i][j] * x.value +
12         gradient[i] * x.gradient[j] +
13         gradient[j] * x.gradient[i] +
14         value * x.Hessian[i][j];
15     }
16   }
17   return temp;
18 }
```

Listing 4.4: Our modified multiplication operator

Our multiplication operator has doubled in size in terms of lines of code, but the added lines are not particularly complicated. On line 8 we start to compute the values of the Hessian matrix according to (4.2). This is simply done by nesting two for-loops and calculating element f_{ij} in the inner loop.

Even though we have made some changes that significantly improves the usefulness of our ADouble class, the actual way to use the class in our code

has not changed. Consider the function

$$f(x) = x_0^3 + x_1^3 + x_2^3 + x_0 \cdot x_1^2 + x_1 \cdot x_2^2 + x_0 \cdot x_2^2 \tag{4.3}$$

its gradient

$$\begin{aligned} \nabla f(x) = & \left(\tfrac{\partial f}{\partial x_0}, \tfrac{\partial f}{\partial x_1}, \tfrac{\partial f}{\partial x_2}\right)^T \\ = & \left(3x_0^2 + x_1^2 + x_2^2, 3x_1^2 + 2x_0x_1 + x_2^2, 3x_2^2 + 2x_1x_2 + 2x_0x_2\right)^T \end{aligned}$$

and the Hessian

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_0^2} & \frac{\partial^2 f}{\partial x_0 \partial x_1} & \frac{\partial^2 f}{\partial x_0 \partial x_2} \\ \frac{\partial^2 f}{\partial x_1, \partial x_0} & \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2, \partial x_0} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 6x_0 & 2x_1 & 2x_2 \\ 2x_1 & 6x_1 + 2x_0 & 2x_2 \\ 2x_2 & 2x_2 & 6x_2 + 2x_1 + 2x_0 \end{bmatrix}$$

We want to compute the function value, gradient and Hessian of function (**??**) in the point $x = (x_0, x_1, x_2) = (1, 2, 3)$. This function value is

$$f(x) = 1^3 + 2^3 + 3^3 + 1 \cdot 2^2 + 2 \cdot 3^2 + 1 \cdot 3^2 = 67, \tag{4.4}$$

the gradient

$$\begin{aligned} \nabla f(x) = & \left(3 \cdot 1^2 + 2^2 + 3^2, 3 \cdot 2^2 + 2 \cdot 1 \cdot 2 + 3^2, 3 \cdot 3^2 + 2 \cdot 2 \cdot 3 + 2 \cdot 1 \cdot 3\right)^T \\ = & \left(16, 25, 45\right)^T \end{aligned}$$

and the Hessian

$$\nabla^2 f(x) = \begin{bmatrix} 6 \cdot 1 & 2 \cdot 2 & 2 \cdot 3 \\ 2 \cdot 2 & 6 \cdot 2 + 2 \cdot 1 & 2 \cdot 3 \\ 2 \cdot 3 & 2 \cdot 3 & 6 \cdot 3 + 2 \cdot 2 + 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 6 & 4 & 6 \\ 4 & 14 & 6 \\ 6 & 6 & 24 \end{bmatrix}.$$

Consider the following code to compute the function value, gradient and Hessian of function (4.3)

```
1  #include <iostream>
2  #include "ADouble.h"
3
4  int main() {
5    const int num_vars = 3;
6    ADouble x0(num_vars, 1.0, 0);
7    ADouble x1(num_vars, 2.0, 1);
8    ADouble x2(num_vars, 3.0, 2);
9
10   ADouble f(num_vars, 0.0,0);
11   f = x0*x0*x0 + x1*x1*x1 + x2*x2*x2 +
12     x0*x1*x1 + x1*x2*x2 + x0*x2*x2;
13   std::cout << f;
14   return 0;
15 }
```

Listing 4.5: Computing the Hessian using our ADouble class

Just like we did previously, we start by defining how many variables our function consists of. The next order of business is to declare and initialise our variables so that we evaluate f at the desired point. This is done on line 6 - 8. Finally, we define the function itself on line 11. The resulting output from Listing 4.5 is

```
Value: 67
Gradient: [ 16  25  45 ]
Hessian:
6
4  14
6  6  24
```

The values produced by our ADouble class is the same as the values we got when we calculated the derivatives by hand, which is quite impressive for little more than one hundred lines of code. To do computations on some real life problems we would of course need to implement other operators, perhaps also overload functions such as sin() and cos(), but the procedure to do so differs little from what we have already seen.

4.3 Extending ADouble to the third derivative

As the chapter heading implies, we are interested in more than the second partial derivatives. We want to compute the third derivative as well. The collection of third order partial derivatives, henceforth called the *Tensor*, is

not quite so easily visualised as the gradient or the Hessian. Recall that the gradient is defined as a vector with each element defined as

$$\frac{\partial f}{\partial x_i}$$

and the second partial derivative is a matrix with each element defined as

$$\frac{\partial^2 f}{\partial x_i \partial x_j}$$

Consider the Tensor as the n times n times n cube with each element, or 'block' as it were, defined as

$$\frac{\partial^3 f}{\partial x_i \partial x_j \partial x_k}$$

We will use $\nabla^3 f(x)$ to denote the Tensor of the function $f(x)$ where $f \colon \mathbb{R}^n \to \mathbb{R}$.

We will again turn to the trusty old multiplication operator to show us how to compute the elements of the Tensor. Consider again the function

$$f(x) = u \cdot v$$

Element ijk of the Tensor is calculated using

$$f_{ijk} = \frac{\partial}{\partial x_k} f_{ij} = \frac{\partial}{\partial x_k}(u_{ij} \cdot v) + \frac{\partial}{\partial x_k}(u_i \cdot v_j) + \frac{\partial}{\partial x_k}(u_j \cdot v_i) + \frac{\partial}{\partial x_k}(u \cdot v_{ij}) \quad (4.5)$$

The worrying development we began to see regarding computational complexity and storage requirements of the Hessian becomes even worse for the Tensor. Based on what we have seen so far, the Tensor will require a ridiculous n^3 doubles of memory. Not to mention how many operations we need to actually compute the Tensor. But we will again throw caution to the wind and continue bravely on our quest to expand our naive implementation.

Consider the new header definition for our ADouble class

```
1 #ifndef ADOUBLE_H_
2 #define ADOUBLE_H_
3
4 class ADouble {
5 private:
6   double value;
7   double * gradient;
```

```
 8    double ** Hessian;
 9    double *** Tensor;
10    int num_vars;
11
12  public:
13    ADouble(int num_vars, double v, int diffindex);
14    virtual ~ADouble();
15
16    ADouble operator+ (const ADouble& x);
17    ADouble operator* (const ADouble& x);
18    ADouble& operator= (const ADouble& x);
19    friend std::ostream& operator<< (std::ostream& s, const ADouble
          &);
20  };
21  #endif // ADOUBLE_H_
```

Listing 4.6: Header file of our ADouble class

Only one change has been made from Listing 4.1 with the declaration of the Tensor on line number 9.

We will also need to make further changes to the source file. The constructor needs to be modified to allocate the needed memory to store the Tensor. As was the case with the Hessian, the Tensor will also be initialized with all its elements set to zero. The destructor will free the memory when we no longer require it, and the operator will be modified to perform calculations on the Tensor.

Let us first have a look at our new ADouble class constructor

```
1  ADouble::ADouble(int nvars, double v, int diffindex) {
2    value = v;
3    num_vars = nvars;
4    gradient = new double[num_vars];
5    for(int i=0; i<num_vars; i++) {
6      gradient[i] = 0;
7    }
8    gradient[diffindex] = 1;
9    Hessian = new double*[num_vars];
10   for(int i=0; i<num_vars; i++) {
11     Hessian[i] = new double[num_vars];
12   }
13   for(int i=0; i<num_vars; i++) {
14     for(int j=0; j<num_vars; j++) {
15       Hessian[i][j] = 0;
16     }
17   }
18   Tensor = new double ** [num_vars];
```

```
19    for(int i=0; i<num_vars; i++) {
20      Tensor[i] = new double*[num_vars];
21      for(int j=0; j<num_vars; j++) {
22        Tensor[i][j] = new double[num_vars];
23      }
24    }
25    for(int i=0; i<num_vars; i++) {
26      for(int j=0; j<num_vars; j++) {
27        for(int k=0; k<num_vars; k++) {
28          Tensor[i][j][k] = 0;
29        }
30      }
31    }
32  }
```

Listing 4.7: A constructor for our ADouble class

The constructor i Listing 4.7 has nearly doubled in size since Listing 4.2, but the changes are not particularly complicated. Lines 18 through 24 handles the allocation of the n x n x n sized Tensor, the 25 through 32 simply initialises the Tensor to 0. The destructor, which will not be listed, is modified accordingly to free the memory when the object's lifetime expires.

Not surprisingly, changes are also made to the operators. Here illustrated by the following Listing for the multiplication operator.

```
1  ADouble ADouble::operator*(const ADouble& x) {
2    ADouble temp(x.num_vars, 0.0, 0);
3    temp.value = value * x.value;
4    for(int i=0; i<x.num_vars; i++) {
5      temp.gradient[i] = (gradient[i] * x.value) + (value * x.
         gradient[i]);
6    }
7    for(int i=0; i<x.num_vars; i++) {
8      for(int j=0; j<x.num_vars; j++) {
9        temp.Hessian[i][j] =
10         Hessian[i][j] * x.value +
11         gradient[i] * x.gradient[j] +
12         gradient[j] * x.gradient[i] +
13         value * x.Hessian[i][j];
14     }
15   }
16   for(int i=0; i<x.num_vars; i++) {
17     for(int j=0; j<x.num_vars; j++) {
18       for(int k=0; k<x.num_vars; k++) {
19         temp.Tensor[i][j][k] =
20           Tensor[i][j][k] * x.value +
21           Hessian[i][k] * x.gradient[j] +
22           Hessian[j][k] * x.gradient[i] +
```

```
23              gradient[k] * x.Hessian[i][j] +
24              Hessian[i][j] * x.gradient[k] +
25              gradient[i] * x.Hessian[j][k] +
26              gradient[j] * x.Hessian[i][k] +
27              value * x.Tensor[i][j][k];
28          }
29      }
30    }
31    return temp;
32  }
```

Listing 4.8: A multiplication operator for our ADouble class

Just as the case was in listing 4.4, our multiplication operator has doubled in size compared to its previous version. On lines 16 through 30 we compute the Tensor. There is not any more magic involved than it was when we calculated the Hessian. All that is needed is three nested for-loops, the inner loop calculating each and every Tensor element ijk according to 4.4.

We use the class in exactly the same manner we did in Listing 4.5 to compute the Tensor in addition to the the function value, gradient and Hesian at any given point.

Chapter 5

Exploiting structure and sparsity

5.1 Partially separable functions

Consider a partially separable function on the form

$$f(x) = \sum_{k=1}^{m} \phi_k(x_k, x_{k+d_1}, x_{k+d_2}, \ldots, x_{k+d_\beta}) \tag{5.1}$$

where $d_i \in \mathbb{D}$

Theorem 1. *[22] If f is three times continuously differentiable then the Hessian matrix has the diagonals*

$$\mathbb{I} = \{i - j \mid i \in \mathbb{D}, j \in \mathbb{D}\} \tag{5.2}$$

and the third derivative has the diagonals

$$\mathbb{III} = \{(i, j) \mid i \in \mathbb{I}, j \in \mathbb{I}, i - j \in \mathbb{I}\} \tag{5.3}$$

Proof. We only need to show that $\mathbb{I}$ given in (5.2) are the diagonals of the Hessian matrix of the elemental functions. Consider

$$\phi_k(x_k, x_{k+d_1}, x_{k+d_2}, \ldots, x_{k+d_\beta})$$

then

$$\frac{\partial^2}{\partial x_{k+i} \partial x_{k+j}} \phi_k \quad i \in \mathbb{D}, j \in \mathbb{D}$$

are the only nonzero elements of the element function ϕ_k. Element $(k+i, k+j)$ in the Hessian matrix is in diagonal $k + i - k - j = i - j$. The diagonals of the Hessian matrix are thus given by (5.2). The second part of the theorem follows from the observation that $(i, j) \in \mathbb{III}$ if and only if $i \in \mathbb{I}, j \in \mathbb{I}, i - j \in \mathbb{I}$. □

When we now consider the class of problems described in (5.1), we have the luxury of a priori information about the structure of the elements of the second and third partial derivatives.

5.2 Storing a sparse Hessian

Very large sparse matrices often appear in science or engineering when solving partial differential equations, meaning the matrices has many nonzero elements. It is difficult to specify exactly what this 'small number' is, but from a practical point of view, a matrix can be called sparse if it has so many zero elements that it is worth to inspect its structure and use appropriate methods to save storage and number of operations.

The Hessian matrix of a function on the form given in (5.1) produces diagonal banded matrices, quite often with a great many zero elements, and is therefore a good example of where appropriate datastructures and methods can be to utilised to great effect.

The basic idea when storing sparse matrices is to store the non-zero elements, and perhaps also a limited number of zeros. This requires a scheme for knowing where the elements fit into the full matrix. There are many methods for storing the data of sparse matrices. See [4] and [9] for details on formats such as compressed row storage (CRS), compressed diagonal storage (CDS) and jagged diagonal storage (JDS). CDS is also used in commercial products from IMB [11] and Intel [12].

See also [7] for a comparison on the efficiency of CRS and Jagged CRS. Even though [7] uses Java, the indication on performance should translate to this thesis because we use arrays of arrays here as well. This means that if need be, we are able to manipulate the rows independently without upsetting the structure as would be necessary if we used CRS.

Seeing that the class of functions described in (5.1) produces diagonal Hessians, a natural storage scheme would be one that utilises the diagonal nature of the matrix. For this reason we use Jagged Compressed Diagonal Storage (JCDS) [8], a storage scheme which uses the technique introduced with Jagged Sparse Array. JCDS is also implemented by Bjørn-Ove Heimsund in the package Matrix Toolkits for Java (MTJ).

The Hessian is stored by jagged compressed diagonal storage when we store:

- A one-dimensional integer array *Iptr* the size of the number of nonzero

Hessian diagonals. This array will contain the row indexes for each diagonal.

- A two-dimensional floating point array containing the numerical values of the Hessian matrix.

Consider the following header definition for a JCDS class

```
1  #ifndef JCDS_H_
2  #define JCDS_H_
3
4  #include "Vector.h"
5
6  class JCDS {
7  private:
8    double ** data;
9    int * Iptr;
10   int num_diags;
11   int num_vars;
12
13 public:
14   JCDS(int ndiags, int nvars, int indexes[]);
15   JCDS(int ndiags, int nvars, int indexes[], double value);
16   JCDS(const JCDS& Hessian);
17   virtual ~JCDS();
18   //Get/set methods omitted
19   ...
20   Vector operator * (const Vector& x);
21   JCDS operator * (const JCDS& x);
22   JCDS operator + (const JCDS& x);
23   JCDS operator = (const JCDS& x);
24   friend std::ostream& operator<<(std::ostream& s, const JCDS& c)
         ;
25 };
26
27 #endif /*CDS_H_*/
```

Listing 5.1: Header definition for the JCDS class

On line 8 we declare our two-dimensional floating point array. This array will store the diagonals of the Hessian matrix. Our one-dimensional integer array of row indexes for each diagonal is declared on line 9. It is also prudent to define a handful of operators for our JCDS class. Matrix-vector products, matrix addition and multiplication are all important operations if our class is to see any practical use. These operators will not be described further. It is also convenient to be able to print the class to standard output, which is done by overloading the output stream operator on line 34.

Consider a constructor for the JCDS class:

```
1  JCDS::JCDS(int ndiags, int nvars, int indexes[]) {
2    num_diags = ndiags;
3    num_vars = nvars;
4
5    Iptr = new int[num_diags];
6    for(int i=0; i<num_diags; i++) {
7      Iptr[i] = indexes[i];
8    }
9
10   data = new double*[num_diags];
11   for(int i=0; i<num_diags; i++) {
12     int size = num_vars - Iptr[i];
13     data[i] = new double[size];
14   }
15
16   for(int i=0; i<num_diags; i++) {
17     int size = num_vars - Iptr[i];
18     for(int j=0; j<size; j++) {
19       data[i][j] = 0;
20     }
21   }
22 }
```

Listing 5.2: A constructor for the JCDS class

In order to create an instance of the JCDS class we need the number of independent variables, the number of diagonals in the Hessian, and the row-indexes of the diagonals. First the Iptr array is created, then the array of arrays is allocated, and finally the elements are initialized to zero.

Consider the Hessian matrix: [1]

$$H = \begin{bmatrix} 2 & 0 & 6 & 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 1 & 4 & 0 & 4 \\ 6 & 0 & 3 & 0 & 3 & 1 & 0 \\ 3 & 1 & 0 & 5 & 0 & 2 & 2 \\ 0 & 4 & 3 & 0 & 7 & 0 & 7 \\ 4 & 0 & 1 & 2 & 0 & 8 & 0 \\ 0 & 4 & 0 & 2 & 7 & 0 & 2 \end{bmatrix}. \tag{5.4}$$

This matrix stored with JCDS will have the following Iptr:

$$Iptr = \{0, 2, 3, 5\}$$

and the elements will be stored as the array of arrays

$$data = \{\{2, 4, 3, 5, 7, 8, 2\}, \{6, 1, 3, 2, 7\}, \{3, 4, 1, 2\}\{4, 4\}\}$$

[1] Numerical values are for illustrative purposes.

5.3 The symmetry and sparsity of the Tensor

In order to compute efficiently with the Tensor, we have got to utilise its symmetry and sparsity. The sparsity of the Tensor is induced by the sparsity of the Hessian, and because the function (5.1) is three times continually differentiable we can utilise the super-symmetry. Just as the case is with the second partial derivative, the partial derivatives of the function $f\colon \mathbb{R}^n \to \mathbb{R}$ at any given point in $\mathbb{R}^n$ commute. Meaning that any combination of the indices i, j and k at any given point produce the same partial derivatives. For instance

$$\frac{\partial^3 f}{\partial x_i \partial x_j \partial x_k} x = \frac{\partial^3 f}{\partial x_k \partial x_i \partial x_j} x \tag{5.5}$$

We will again turn to [8] to provide us with an appropriate datastructure to store the induced Tensor, namely jagged compressed diagonal storage for induced Tensor (JCDST). The Tensor is stored by JCDST when we store:

- A one-dimensional integer array *Iptr* the size of the number of nonzero Hessian diagonals. This array contains the row indexes for each nonzero diagonal in the Hessian.
- A two-dimensional sparse matrix *IIptr* the size of the number of nonzero diagonals in the Tensor, which will store the j indexes for each nonzero diagonal in the Tensor. This is a pointer structure that gives the start indexes for each Tensor diagonal.
- A three-dimensional floating point array for storing the numerical values in the Tensor.

Consider the jagged compressed diagonal storage for induced tensors induced from the matrix in (5.4)

$$Iptr = \{0, 2, 3, 5\}$$

IIptr will contain

$$IIptr = \{0, \{0, 2\}, \{0, 3\}, \{0, 2, 3, 5\}\}$$

and the elements will be stored as [2]

$$\begin{aligned} data = \{&\{2, 3, 11, 12, 16, 19, 24\},\\ &\{3, 6, 7, 10, 15\}, \{4, 6, 9, 13, 15\},\\ &\{7, 9, 10, 11, 15\}, \{9, 10, 13, 14, 17\},\\ &\{11, 13\}, \{15, 20\}, \{31, 40\}, \{36, 42\}\} \end{aligned}$$

[2] Numerical values are for illustrative purposes.

Consider the header definition for our JCDST class

```
1  #include "Vector.h"
2  #include "JCDS.h"
3
4  class JCDST {
5  private:
6    int num_vars;
7    int num_diags;
8    int * Tsizes;
9    int * Iptr;
10   int ** IIptr;
11   int * IIsizes;
12   double *** Tval;
13 public:
14   JCDST(int ndiags, int n, int Iptr[]);
15   JCDST(int ndiags, int n, int Iptr[], double value);
16   JCDST(const JCDST& x);
17   virtual ~JCDST();
18
19   // Get/set methods omitted
20   ...
21
22   JCDS operator * (const Vector& x);
23   friend JCDS operator * (const Vector& p, const JCDST& T);
24   JCDST operator = (const JCDST& x);
25   friend std::ostream& operator<<(std::ostream &s, const JCDST &c
         );
26 };
```

Listing 5.3: Header definition for the JCDST class

The data members are all defined on line 6 - 12, and we provide our usual collection of constructors as well as a destructor.

As was the case with the JCDS header in Listing 5.1, we define a few operators here as well. Most notably the multiplication operator, which says that a vector times a JCDST Tensor produces a JCDS matrix.

Consider a constructor for the JCDST class

```
1  JCDST::JCDST(int ndiags, int n, int sizes[]) {
2    int size = 0;
3    num_diags = ndiags;
4    num_vars = n;
5    Iptr = new int[num_diags];
6    for(int i=0; i<num_diags; i++)
7      Iptr[i] = sizes[i];
8    IIsizes = new int[num_diags];
9    IIptr = new int * [num_diags];
10   for(int i=0; i<num_diags; i++) {
```

```
11      size = 0;
12      for(int j=0; j<=i; j++) {
13        int search_for = Iptr[i] - Iptr[i-j];
14        for(int k=0; k<num_diags; k++) {
15          if(Iptr[k]==search_for) {
16            size++;
17          }
18        }
19      }
20      IIptr[i] = new int[size];
21      IIsizes[i] = size;
22      int index = 0;
23      for(int j=0; j<=i; j++) {
24        int search_for = Iptr[i] - Iptr[i-j];
25        for(int k=0; k<num_diags; k++) {
26          if(Iptr[k]==search_for) {
27            IIptr[i][index] = Iptr[k];
28            index++;
29          }
30        }
31      }
32    }
33    Tsizes = new int[num_diags];
34    Tval = new double ** [num_diags];
35    for(int i=0; i<num_diags; i++) {
36      Tval[i] = new double * [IIsizes[i]];
37      for(int j=0; j<IIsizes[i]; j++) {
38        int tsize = n - Iptr[i];
39        Tval[i][j] = new double [tsize];
40        Tsizes[i] = tsize;
41      }
42    }
43    for(int i=0; i<num_diags; i++) {
44    int size = IIsizes[i];
45      for(int j=0; j<size; j++) {
46        int tsize = Tsizes[i];
47        for(int k=0; k<tsize; k++) {
48          Tval[i][j][k] = 0;
49        }
50      }
51    }
52  }
```

Listing 5.4: A constructor for the JCDST class

5.4 Using JCDS and JCDST in our ADouble class

Consider the new header definition for our ADouble class:

```
1  #ifndef ADOUBLE_H_
2  #define ADOUBLE_H_
3
4  #include "JCDST.h"
5  #include "JCDS.h"
6
7  class ADouble {
8  private:
9    int diff_index_;
10   JCDST * Tensor;
11   JCDS * Hessian;
12   int num_diags_;
13   double * gradient;
14   int num_vars_;
15   int * Iptr_;
16   double value;
17
18 public:
19   ADouble(int num_vars, double v, int diffindex, int Iptr[],
20           int num_diags, bool is_static);
21   ADouble(int num_vars, double v, int diffindex, int Iptr[],
22           int num_diags);
23   virtual ~ADouble();
24
25   ADouble operator+ (const ADouble& x);
26   ADouble operator+ (const double& x);
27   friend ADouble operator+ (const double& x, const ADouble& y);
28
29   ADouble operator- (const ADouble& x);
30   ADouble operator- (const double& x);
31   friend ADouble operator- (const double& x, const ADouble& y);
32
33   ADouble operator* (const ADouble& x);
34   ADouble operator* (const double& x);
35   friend ADouble operator* (const double& x, const ADouble& y);
36
37   ADouble& operator= (const ADouble& x);
38
39   friend std::ostream& operator<< (std::ostream& s, const ADouble
         &);
40 };
41 #endif // ADOUBLE_H_
```

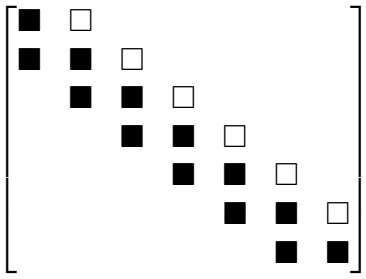
Listing 5.5: Header definition for the ADouble class

Note that the ADouble class now has two different constructors. The constructor with the boolean variable is_static is called every time we initialise an independent variable. This is because for any independent variable, $\frac{d}{dx}x = 1$ and any higher derivatives will necessarily become zero. This enables us to initialise all our independent variables with a NULL-pointer to an instance of the JCDS and JCDST class, which will save us from allocating a lot of unnecessary memory.

Consider the function

$$f(x) = \sum_{i=1}^{n-1} \alpha(x_i - x_{i-1}^2)^2 \cdot (1 - x_{i-1})^2 \tag{5.6}$$

The function in Eq 5.6 will produce a Hessian with the following structure:



The empty squares indicate elements that will be stored implicitly due to the Hessian's symmetry over the main diagonal. With our JCDS class we will then need to store the main diagonal and the subdiagonal, totalling $n + n - 1$ doubles for each dependant variable. Suppose that $n = 1000$ and our independent variables are stored in an array. Consider the following code excerpt:

```
1 for(int i=1; i<n; i+=2) {
2   f = f + (alpha *
3     ((x[i] - x[i-1] * x[i-1]) *
4     (x[i] - x[i-1] * x[i-1]))) +
5     ((1 - x[i-1]) * (1 - x[i-1]));
6 }
```

Listing 5.6: Implementing (5.6)

Unless the compiler is very good at optimising code, dependant variables will be created by all the elementary operations required to compute the expression for each execution of the for-loop. These dependant variables

are all instances of the ADouble class. When the scope of the generated dependant variables expires, in this case when execution reaches the equal-sign on line 2, the destructor of all the dependant variables will be called and the allocated memory will be freed.

Counting the number of additions, multiplications and subtractions in Listing 5.6, let us say the compiler creates 10 dependant variables for each execution of the for-loop. Without differentiating between independent and dependant variables, we would have to allocate $(n + n - 1) = 1999$ doubles of memory for **each** of the n independent variables just to store the Hessian. In contrast, we only need to allocate $(n + n - 1)$ doubles of memory for each the 10 dependant variables created by the compiler.

We need to make several changes to our arithmetic operators in order to use the JCDS an JCDST datastructures. Consider the addition operator for the ADouble class:

```
1  ADouble ADouble::operator+ (const ADouble& x) {
2    ADouble temp(x.num_vars_, 0.0, -1, Iptr_, num_diags_);
3    temp.value = value + x.value;
4    for(int i=0; i<x.num_vars_; i++) {
5      temp.gradient[i] = gradient[i] + x.gradient[i];
6    }
7    double aval;
8    double hessij;
9    double xhessij;
10   int size;
11   if((Hessian!=NULL) && (x.Hessian!=NULL)) {
12     for(int i=0; i<num_diags_; i++) {
13       size = num_vars_ - Iptr_[i];
14       for(int j=0; j<size; j++) {
15         hessij = Hessian->getElement(i,j);
16         xhessij = x.Hessian->getElement(i,j);
17         aval = hessij + xhessij;
18         temp.Hessian->setElement(i,j,aval);
19       }
20     }
21   } else if (Hessian!=NULL) {
22     for(int i=0; i<num_diags_; i++) {
23       size = num_vars_ - Iptr_[i];
24       for(int j=0; j<size; j++) {
25         aval = Hessian->getElement(i,j);
26         temp.Hessian->setElement(i,j,aval);
27       }
28     }
29
30   } else if (x.Hessian!=NULL) {
31     for(int i=0; i<num_diags_; i++) {
32       size = num_vars_ - Iptr_[i];
33       for(int j=0; j<size; j++) {
34         aval = x.Hessian->getElement(i,j);
35         temp.Hessian->setElement(i,j,aval);
36       }
37     }
38
39   }
40   double tijk;
41   double xtijk;
42   if((Tensor!=NULL) && (x.Tensor!=NULL)) {
43     for(int i=0; i<num_diags_; i++) {
```

```
44|       for(int j=0; j<temp.Tensor->getIIptrSize(i); j++) {
45|         for(int k=0; k< temp.Tensor->getTsize(i); k++) {
46|           tijk = Tensor->getElement(i,j,k);
47|           xtijk = x.Tensor->getElement(i,j,k);
48|           aval = tijk + xtijk;
49|           temp.Tensor->setElement(i,j,k,aval);
50|         }
51|       }
52|     }
53|   } else if (Tensor!=NULL) {
54|     for(int i=0; i<num_diags_; i++) {
55|       for(int j=0; j<temp.Tensor->getIIptrSize(i); j++) {
56|         for(int k=0; k< temp.Tensor->getTsize(i); k++) {
57|           aval = Tensor->getElement(i,j,k);
58|           temp.Tensor->setElement(i,j,k,aval);
59|         }
60|       }
61|     }
62|   } else if (x.Tensor!=NULL) {
63|     for(int i=0; i<num_diags_; i++) {
64|       for(int j=0; j<temp.Tensor->getIIptrSize(i); j++) {
65|         for(int k=0; k< temp.Tensor->getTsize(i); k++) {
66|           aval = x.Tensor->getElement(i,j,k);
67|           temp.Tensor->setElement(i,j,k,aval);
68|         }
69|       }
70|     }
71|
72|   }
73|   return temp;
74| }
```

Listing 5.7: An addition operator for the ADouble class

We start by calculating the gradient in the usual fashion. On lines 11 - 39 we check if any of the arguments to the operator has a NULL-pointer to the Hessian. If that is the case we do not have to calculate anything, because, as outlined previously, the Hessian of independent variables contains only zero elements. This is also the case for the Tensor calculations on lines 42 - 74.

The complexity of the code grows quite rapidly when dealing with higher order derivatives. Recall that our we successfully calculated a function's Hessian in chapter 4.2 with little more than one hundred lines of code. This is in stark contrast to the case we have now, with a simple addition operator counting 74 lines of code.

Chapter 6

Benchmarking

6.1 Floating point operations as a measure of efficiency

A computer program's runtime, or duration of the programs execution, is often used to measure performance. Typically the runtime of the program is measured several times, with the average or weighted average runtime determining the program's efficiency. In our case this approach is less than ideal. The code in our ADouble class and data structures is written for readability rather than speed. A much more accurate method to measure our code's efficiency is to count the number of floating point operations done by our ADouble class. This is implemented by introducing a static class member.

Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another. In fact, static members have the same properties as global variables but they enjoy class scope. Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class.

It is prudent to differentiate between the floating point operations needed to calculate the function value, gradient, Hessian and Tensor. This is done by simply declaring four different static data members for our ADouble class. Then all that is needed is to increment each of these class members by one each time a floating point operation is done in the ADouble class to produce either of the derivatives.

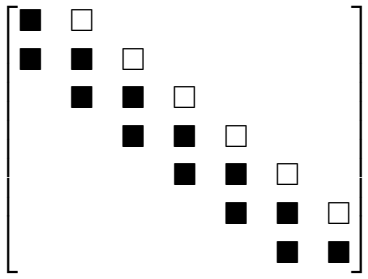
6.2 Test functions

We will consider the following test functions:

Extended Rosenbrock

$$f(x) = \sum_{i=1}^{n-1} \alpha \cdot (x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$$

This function produces a Hessian with the following structure for $n = 7$:



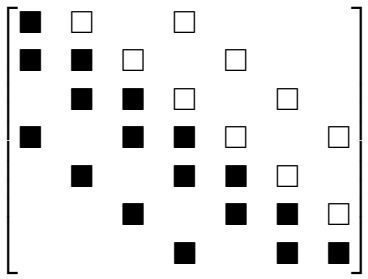
The elements represented by white squares are not stored explicitly due to the symmetry of the Hessian. The black squares are the elements which we store in the JCDS class.

Chained Singular function

$$f(x) = \sum_{i \in \mathbb{J}} [(x_i + 10_{x_{i+1}})^2 + 5(x_{i+2} - x_{i+3})^2 + (x_{i+1} - 2x_{i+2})^4 + 10(x_i - 10x_{i+3})^4]$$

where n is a multiple of 4 and $J = \{0, 2, 4, \cdots, n-3\}$.

This function produces a Hessian with the following structure for $n = 7$.



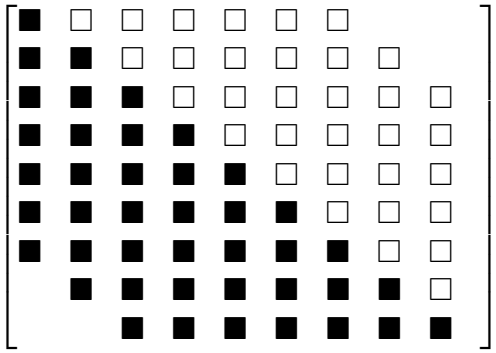
Just as the case is in the Rosenbrock function above, the elements represented by the black squares are the only elements which are stored explicitly.

Broyden Banded

$$f(x) = \sum_{i=0}^{n-1} \left| (2 + 5x_i^2)x_i + \sum_{j \in J_i} x_j(1 + x_j) \right|^p$$

where $p = 4$ and $J_i = \{j : max(1, i-5) \leq j \leq min(n-1, i+1)\}$.

This function produces a Hessian with the following structure for $n = 9$.



Again, black squares indicate elements which are stored in the JCDS class and the white squares are stored implicitly by the symmetry of the Hessian.

6.3 Numerical results

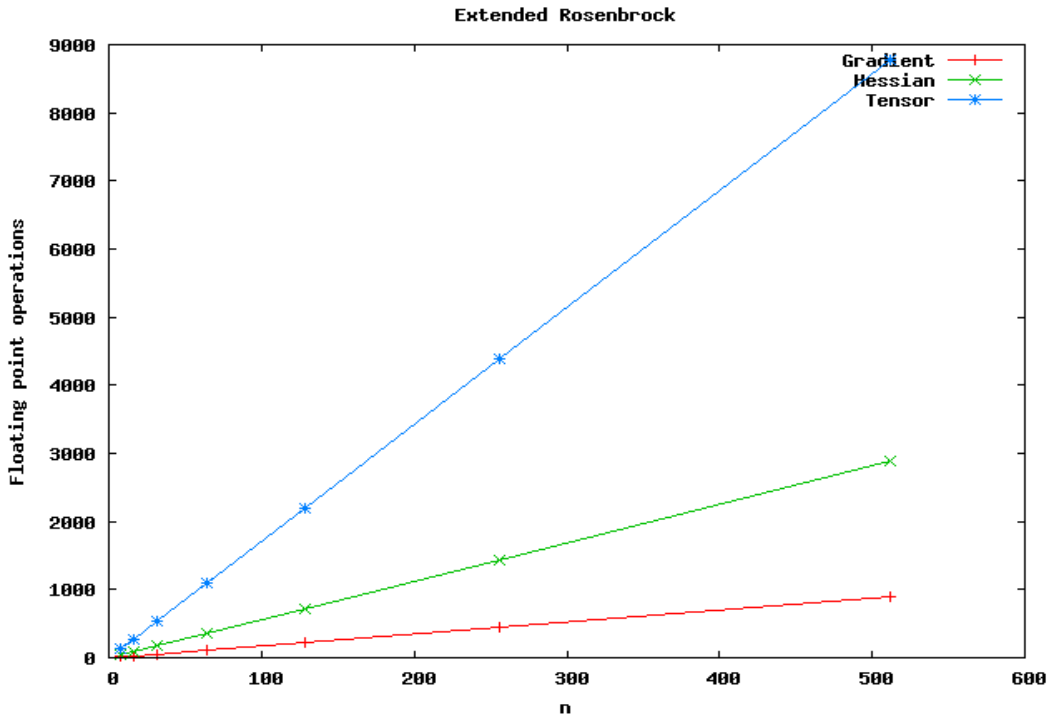
Each of the functions in section 6.2 above are implemented using our ADouble class to compute the derivatives. The data structures used to store the Hessian and Tensor is JCDS and JCDST.

The following tables displays the number of floating point operations needed to calculate the function value, gradient, Hessian and Tensor for each of the test functions above.

6.3.1 Extended Rosenbrock

Extended Rosenbrock ($\alpha = 100$)				
n	function value	gradient	Hessian	Tensor
8	44	608	1860	5544
16	88	2432	7688	23184
32	176	9728	31248	94752
64	352	38912	125984	383040
128	704	155648	505920	$1.54022 \cdot 10^6$
256	1408	622592	$2.02765 \cdot 10^6$	$6.17702 \cdot 10^6$
512	2816	$2.49037 \cdot 10^6$	$8.11853 \cdot 10^6$	$2.47404 \cdot 10^7$

From the table above, we can immediately notice that for each time we double n, we quadruple the floating point operations needed to calculate each of the derivatives. The gradient for this problem is relatively expensive to calculate with its $9.5n^2$ floating point operations, but it is very interesting to see how the higher derivatives grow if we relate them to the operations needed for the function value.



Here we have plotted the gradient, Hessian, and Tensor, all of which are scaled by the number of floating point operations needed to calculate the function value. The number of operations needed to calculate the function value grows with n and so does not alter the plots.

These relationships are denoted as gradient, Hessian, and Tensor in the plot. With respect to the floating point operations required to calculate the function value, the cost of all the derivatives grows linearly.

It is also interesting to investigate the relations $\frac{gradient}{functionvalue}$, $\frac{Hessian}{gradient}$, and $\frac{Tensor}{Hessian}$, which is given in the following table:

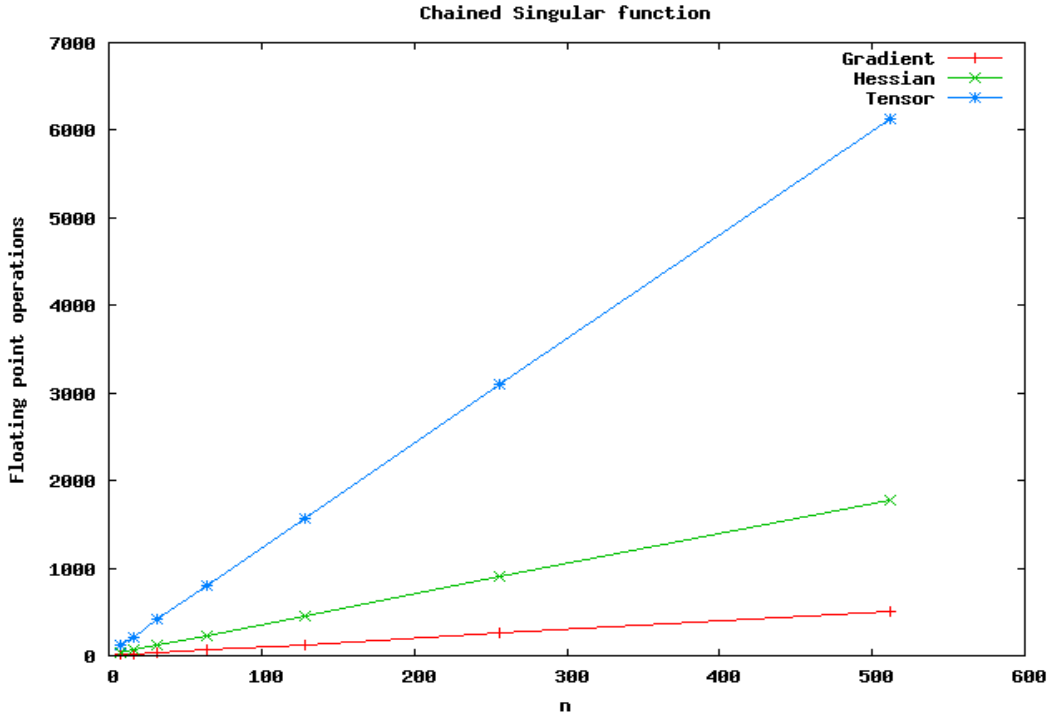
Extended Rosenbrock ($\alpha = 100$)			
n	$\frac{gradient}{functionvalue}$	$\frac{Hessian}{gradient}$	$\frac{Tensor}{Hessian}$
8	13.8182	3.05921	2.98065
16	27.6364	3.16118	3.01561
32	55.2727	3.21217	3.03226
64	110.545	3.23766	3.04039
128	221.091	3.25041	3.0444
256	442.182	3.25678	3.0464
512	884.364	3.25997	3.04739

As we see, the cost of the gradient relative to the function value grows with n, but the Hessian relative to the gradient, and Tensor relative to the Hessian, remains fairly constant.

6.3.2 Chained Singular function

Chained Singular function				
n	function value	gradient	Hessian	Tensor
8	108	1248	3720	12288
16	360	7072	22816	76800
32	900	32032	108376	368640
64	2016	135200	469712	$1.60666 \cdot 10^6$
128	4284	554528	$1.95399 \cdot 10^6$	$6.7031 \cdot 10^6$
256	8856	$2.24515 \cdot 10^6$	$7.96973 \cdot 10^6$	$2.73807 \cdot 10^7$
512	18036	$9.03427 \cdot 10^6$	$3.21906 \cdot 10^7$	$1.10678 \cdot 10^8$

From the table above, we see that the chained singular function behaves in roughly the same manner as the extended Rosenbrock function did earlier. When we double the values for n, the required operations needed to compute the derivatives grows by an approximate factor of 6. Let us investigate how the derivatives grow when we again scale them by the operations needed to calculate the function value.



Like we did with the extended Rosenbrock function, we have plotted the gradient, Hessian, and Tensor, all of which are scaled by the number of floating point operations needed to calculate the function value.

With respect to the floating point operations required to calculate the function value, the cost of all the derivatives grows linearly.

The relations $\frac{gradient}{functionvalue}$, $\frac{Hessian}{gradient}$, and $\frac{Tensor}{Hessian}$ is given in the following table.

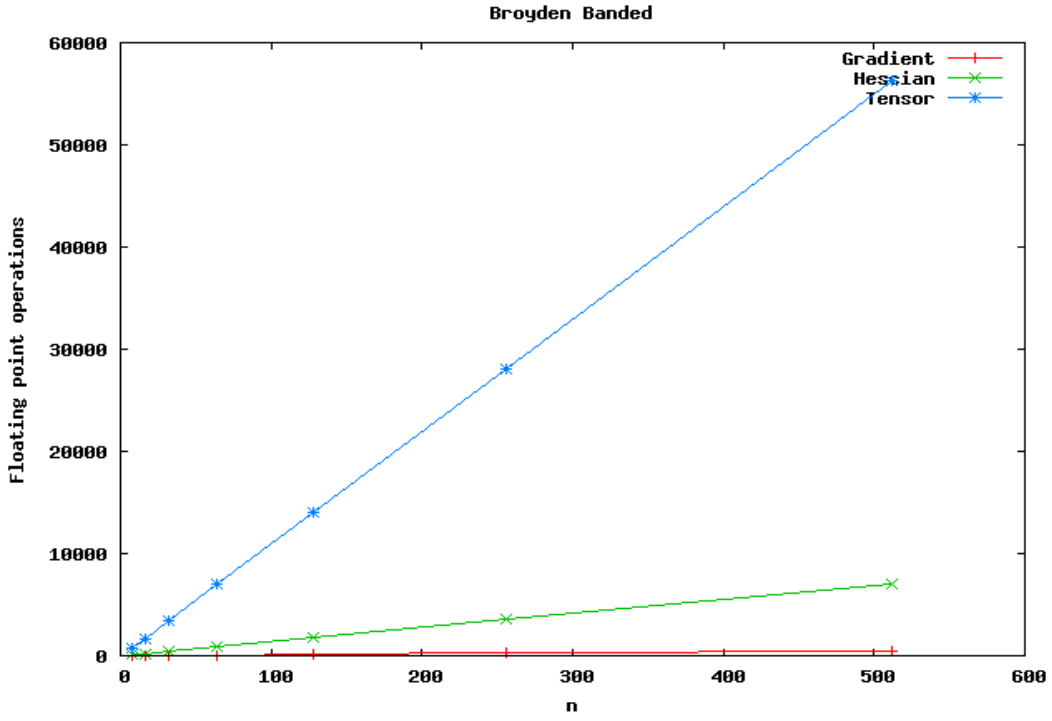
Chained Singular function			
n	$\frac{gradient}{functionvalue}$	$\frac{Hessian}{gradient}$	$\frac{Tensor}{Hessian}$
8	11.5556	2.98077	3.30323
16	19.6444	3.22624	3.36606
32	35.5911	3.38337	3.40149
64	67.0635	3.4742	3.42051
128	129.442	3.5237	3.43047
256	253.518	3.54975	3.43559
512	500.902	3.56317	3.4382

The table above shows us that again, the cost of the gradient relative to the function value grows with n. The Hessian relative to the gradient and Tensor relative to the Hessian does grow, but relatively slowly.

6.3.3 Broyden Banded

Broyden Banded				
n	function value	gradient	Hessian	Tensor
8	192	2368	20160	129024
16	624	12864	136640	989184
32	1536	56896	682304	$5.20397 \cdot 10^6$
64	3408	237120	$3.03251 \cdot 10^6$	$2.36974 \cdot 10^7$
128	7200	966208	$1.27765 \cdot 10^7$	$1.01026 \cdot 10^8$
256	14832	$3.89894 \cdot 10^6$	$5.24469 \cdot 10^7$	$4.17135 \cdot 10^8$
512	30144	$1.56627 \cdot 10^7$	$2.12525 \cdot 10^8$	$1.69525 \cdot 10^9$

This function continues the trend started by the previous two test functions. We see that the derivatives requires roughly six times as many operations when we double the values for n. Let us continue to investigate how the derivatives grow when we again scale them by the operations needed to calculate the function value.



Just as we did in the previous two test functions, the relations $\frac{gradient}{functionvalue}$, $\frac{Hessian}{functionvalue}$, and $\frac{Tensor}{functionvalue}$ are plotted. We see again that with respect to

the floating point operations required to calculate the function value, the cost of all the derivatives grows linearly.

The relations $\frac{gradient}{functionvalue}$, $\frac{Hessian}{gradient}$, and $\frac{Tensor}{Hessian}$ is given in the following table.

Broyden Banded			
n	$\frac{gradient}{functionvalue}$	$\frac{Hessian}{gradient}$	$\frac{Tensor}{Hessian}$
8	12.3333	8.51351	6.4
16	20.6154	10.6219	7.23934
32	37.0417	11.9921	7.62705
64	69.5775	12.7889	7.81445
128	134.196	13.2234	7.90715
256	262.874	13.4516	7.95346
512	519.594	13.5689	7.97668

Again, the operations to compute the gradient relative to that of the function value grows with n, but Hessian relative to the gradient and the Tensor relative to the Hessian grows faster than it did in the previous two test functions. The values of these relations are also significantly higher than they were in both the previous test functions. This can be attributed to the much larger number of nonzero elements in both the Hessian and the Tensor. Furthermore, even if the values of the relations are higher than they were in the previous two functions, they do not grow very rapidly.

Chapter 7

Using ADouble in optimization

Automatic differentiation is used in a variety of fields such as numerical methods, sensitivity analysis and inverse problems. It is also used in other fields not necessarily thought of as science e.g digital movie production [18]. We will illustrate the use of automatic differentiation with an implementation of the conjugate gradient method.

Conjugate gradient methods [21] are very useful for solving large linear systems of equations, and they can also be adopted to solve nolinear optimization problems. The performance of the conjugate gradient method is determined by the distribution of the eigenvalues of the coefficient matrix. By applying a preconditioner to the linear system, we can make this distribution more favorable. However, we will not apply such a preconditioner and focus on a simple implementation of the conjugate gradient method. Consider the following implementation of the conjugate gradient method [1]:

```
1  int CG(Vector& df, JCDS& ddf, int n, Vector& sk1, double tol) {
2    double rho_1 = 0;
3    Vector b = df;
4    JCDS A = ddf;
5    int iter = 0;
6    Vector p(n);
7    Vector x(n);
8    Vector z(n);
9    Vector q(n);
10   Vector r = b - A*x;
11   double brnm2 = b.norm();
12   if(brnm2 == 0.0) {
13     brnm2 = 1.0;
14   }
15   double error = r.norm() / brnm2;
```

[1] Translated to C++ from a Matlab template provided by Geir Gundersen in personal correspondence

```
16    if(error < tol) {
17      sk1 = x;
18      return iter;
19    }
20    double alpha = 0;
21    double beta = 0;
22    while (iter < 50 && error > tol) {
23       z = r;
24       double rho = r*z;
25       if(iter>1) {
26         beta = rho / rho_1;
27         p = z + beta*p;
28       } else {
29         p = z;
30       }
31       q = A*p;
32       alpha = rho / (p*q);
33       x = x + alpha * p;           // update approximation vector
34       r = r - alpha*q;             // compute residual
35       error = r.norm() / brnm2;    // check convergence
36       if(error<=tol) {
37         break;
38       }
39       rho_1 = rho;
40       iter += 1;
41    }
42    sk1 = x;
43    return iter;
44  }
```

Listing 7.1: Conjugate Gradient method in C++

Consider lines 10 and 32. Here we need a vector subtraction, matrix-vector product, and the vector scalar product to be implemented. We will need a class to store vectors with associated member functions such as norms and scalar product. We will also need to implement a matrix-vector product for our JCDS class. These implementation details will not be discussed in this thesis.

The conjugate gradient method in Listing 7.1 will be called from the following function:

```
1  int NewtonsMethodCG(Vector& df, JCDS& ddf, JCDST& dddf, Vector&
       xk, Vector& pk, double alpha, double tol) {
2    int n = xk.getLength();
3    int nh = 0;
4    Vector sk1(n);
5    Vector neg = -df;
6    nh = CG(neg,ddf,n, sk1, tol);
7    pk = sk1;
```

```
8     return nh;
9   }
```

Listing 7.2: Newtons method CG

We will use the following function to minimize:

$$\sum_{i \in \mathbb{I}} \alpha \cdot (x_{i-1} - x_i^2)^2 + (1 - x_i)^2$$

where $\mathbb{I} = \{1, 3, 5, \ldots, n-1\}$ and n is an even number.

We will also need code to generate our independent variables, a vector with the starting point for the algorithm, and a loop to continue the CG algorithm until a stopping criteria is reached. This is all given in the following:

```
1  int main() {
2    double rose_alpha = 6.4;
3    int Iptr[] = {0,1};
4    int num_diags = 2;
5    int n = 10;
6    double f = 0;
7
8    // Generating independent variables
9    ADouble ** x;
10   x = new ADouble * [n];
11   for(int i=0; i<n; i++)
12     x[i] = new ADouble(n, 1.7, i, Iptr, num_diags, true);
13
14   Vector df(n); // gradient
15   JCDS ddf(num_diags, n, Iptr); // Hessian
16   JCDST dddf(num_diags, n, Iptr); // Tensor
17
18   // Starting point
19   Vector xk(n,1.7);
20   Vector pk(n);
21
22   // Calculate initial derivatives
23   f = chainrose(rose_alpha, x, n, df, ddf, dddf);
24
25   Vector df0 = df;
26   double alpha = 1/2;
27   double tol = 0.1;
28   int iter1 = 0;
29   int NH = 0;
30   int nh = 0;
31
32   while(df.norm() >= 1.0e-8 * df0.norm() && iter1 < 20) {
33     nh = NewtonsMethodCG(df, ddf, dddf, xk, pk, alpha, tol);
```

```
34      xk = xk + pk;
35      // Update the independant variables
36      for(int i=0; i<n; i++) {
37        x[i]->setValue(xk(i));
38      }
39      f = chainrose(rose_alpha, x, n, df, ddf, dddf);
40      iter += 1;
41    }
42    std::cout << "Converged in " << iter << " iterations.\n";
43
44    // Memory cleanup
45    for(int i=0; i<n; i++)
46      delete x[i];
47    delete [] x;
48    return 0;
49  }
```

Listing 7.3: Using the CG method

The gradient, Hessian and Tensor defined on line 14-16 will be modified for each call of the chainrose function.

We choose the starting point to be $x = (1.7, 1.7, 1.7, \ldots, 1.7)$ and our tolerance level to be 0.1. With these parameters, our CG implementation minimizes the function $f(x)$ when the norm of the residual is sufficiently small. This is found in 7 iterations with the norm $||\nabla f(x)||_2 = 2.227468438369573 \cdot 10^{-7}$ at the point $x = (0.9999999, 0.9999999, \ldots, 0.9999999)$.

Chapter 8

Conclusions and comments

C++ is a good choice of programming language for implementing automatic differentiation, particularly so if one decides to make an implementation with operator overloading. A multitude of compilers are readily available, and in terms of speed it is no longer a bad choice of programming language compared with Fortran. Advanced features such as expression templates can also be used to improve efficiency.

From the results in chapter 6, even when we only use forward mode, we see that automatic differentiation can indeed be a viable source of derivatives, even for higher order methods. Furthermore, computation of the third derivatives is not of order n^3 as it is stated in [20] and [16], at least not when dealing with partially separable functions. With the growing interest in higher-order methods, especially for problems which require a high degree of accuracy in the solution, calculating the derivatives with the help of AD can be particularly appealing.

The computation of the Tensor in chapter 4 deserves some special mention. It might seem counterproductive to store the Tensor as the full cube with n^3 elements, and to some extent it is. A better solution than the one described in chapter 4 would be to expand on the symmetry of the Hessian to produce a super-symmetric Tensor as outlined in Knuth's Art of Computer Programming, Volume 1.

Chapter 9

Future work

The ADouble class has a lot of room for further development. Exponential and logarithmic functions, trigonometric and inverse trigonometric functions all need to be implemented properly. Furthermore, efficient implementation of matrix-vector and vector-Tensor products, vector norms, and other elements of linear algebra is very important if the ADouble class ever is to see any practical use.

Another intriguing idea lies in how the ADouble class handles memory. As it stands, every independent variable carries with it (possibly) incomplete gradient, Hessian and Tensor objects. Further experiments are needed to determine exactly how the complete derivatives of all orders are computed, and if a single shared gradient, Hessian and Tensor object can be computed upon by all instances of the ADouble class. If this is the case, the memory usage bottleneck can be all but eliminated as all instances of the ADouble class will work with a single piece of shared memory and allocate next to no memory on its own.

In this thesis we have only concerned ourselves with forward mode automatic differentiation. It would be interesting to explore reverse mode, or even a hybrid mode incorporating both forward and reverse mode automatic differentiation.

Some advanced language features of C++ can be of great help to speed up the execution of our code. Expression Templates is a C++ technique for passing expressions as function arguments. The expression can be inlined into the function body, which results in faster and more convenient code than C-style callback functions. The technique of using expression templates can also be used to evaluate vector and matrix expressions in a single pass without temporaries. In preliminary benchmark results, one compiler evaluates vector expressions at 95-99.5% efficiency of hand- coded C using this technique for long vectors [24]. Using expression templates has the added benefit of making

our code independent on the underlying data types. This means that we could use the exact same code to handle any primitive or class datatype.

The old programmer's joke that 'you never finish a program, you just stop working on it' certainly applies to this thesis.

Bibliography

[1] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[2] C. Bendtsen and Ole Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, aug 1996.

[3] Christian H. Bischof, Alan Carle, George F. Corliss, Andreas Griewank, and Paul D. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[4] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, 1994.

[5] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[6] Andreas Griewank, David Juedes, H. Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Institute of Scientific Computing, Technical University Dresden, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[7] G. Gundersen and T. Steihaug. Data structures in java for matrix computations. *Concurrency and Computation: Practice and Experience*, 16(8):799–815, 2004.

[8] Geir Gundersen. *Sparsity in Higher-Order Methods for Unconstrained Optimization.* PhD thesis, University of Bergen, Norway, 2008.

[9] Michael A. Heroux. A proposal for a sparse blas toolkit. Technical report, In preparation, 1992.

[10] F. Kimura I. Ozaki and M. Berz. Higher-order sensitivity analysis of finite element method by automatic differentiation. *Computational Mechanics*, 16(4), 1995.

[11] IBM. Engineering and scientific subroutine library.

[12] Intel. Intel math kernel library.

[13] Max E. Jerrell. Function minimization and automatic differentiation using c++. In *OOPSLA*, pages 169–173, 1989.

[14] Uwe Naumann. Optimal jacobian accumulation is np-complete. *Mathematical Programming*, 112(2):427–441, 2008.

[15] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[16] J. M. Ortega and W. C. Reinholdt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York, NY, 1970.

[17] Valérie Pascual and Laurent Hascoët. TAPENADE for C. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 199–209. Springer, 2008.

[18] Dan Piponi. Automatic differentiation, c++ templates, and photogrammetry. *journal of graphics tools*, 9(4):41–55, 2004.

[19] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.

[20] Werner C. Reinholdt. *Methods for Solving Systems of Nonlinear Equations*. SIAM, 1974.

[21] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[22] Trond Steihaug. personal correspondence, 2008.

[23] Todd L. Veldhuizen. Scientific computing: C++ versus Fortran: C++ has more than caught up. *Dr. Dobb's Journal of Software Tools*, 22(11):34, 36–38, 91, nov 1997.

[24] Todd L. Veldhuizen. C++ templates as partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Tech. Report NS-99-1, pages 13–18. BRICS, 1999.